Free Applicative Functors

Paolo Capriotti
Functional Programming Laboratory
University of Nottingham
pvc@cs.nott.ac.uk

Ambrus Kaposi
Functional Programming Laboratory
University of Nottingham
auk@cs.nott.ac.uk

Abstract

Applicative functors ([9]) are a generalisation of monads. Both allow expressing effectful computations into an otherwise pure language, like Haskell ([8]).
Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values.
We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor.
We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Haskell, Free Applicative Functors

Keywords Applicative Functors, Parametricity, Adjoints

1. Introduction

Free monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded DSLs (see for example [13], where `Free` is called `Term`). In this context, the functor `f` is usually obtained as the coproduct of a number of functors representing "basic operations", and the resulting DSL is the minimal embedded language including those operations.
One problem of the free monad approach is that programs written in a monadic DSL are not amenable to static analysis. It is impossible to examine the structure of a monadic computation without executing it.
In this paper, we show how a similar "free construction" can be realized in the context of applicative functors. In particular, we make the following contributions:

- We give two definitions of *free applicative functor* in Haskell (section 2), and show that they are equivalent (section 5).
- We prove that our definition is correct, in the sense that it really is an applicative functor (section 6), and that it is "free" in a precise sense (section 7).
- We present a number of examples where the use of free applicative functors helps make the code more elegant, removes duplication or enables certain kinds of optimizations which are not possible when using free monads. We describe the differences between expressivity of DSLs using free applicatives and free monads (section 3).
- We compare our definition to other existing implementations of the same idea (section 9).

This paper is aimed at programmers with a working knowledge of Haskell. Familiarity with applicative functors is not required, although it is helpful to understand the motivation behind this work. We make use of category theoretical concepts to justify our definition, but the Haskell code we present can also stand on its own.

1.1 Applicative functors

Applicative functors (also called *idioms*) were first introduced in [9] as a way to provide a lighter notation for monads. They have since been used in a variety of different applications, including efficient parsing (see section 1.4), regular expressions and bidirectional routing.
Applicative functors are defined by the following type class:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The idea is that a value of type `f a` represents an "effectful" computation returning a result of type `a`. The `pure` method creates a trivial computation without any effect, and `( <*> )` allows two computations to be sequenced, by applying a function returned by the first, to the value returned by the second.
Since every monad can be made into an applicative functor in a canonical way, the abundance of monads in the practice of Haskell programming naturally results in a significant number of practically useful applicative functors.
Applicatives not arising from monads, however, are not as widespread, probably because, although it is relatively easy to combine existing applicatives (see for example [10]), techniques to construct new ones have not been thoroughly explored so far.
In this paper we are going to define an applicative functor `FreeA f` for any Haskell functor `f`, thus providing a systematic way to create new applicatives, which can be used for a variety of applications.

The meaning of FreeA f will be clarified in section 7, but for the sake of the following examples, FreeA f can be thought of as the "simplest" applicative functor which can be built using f.

1.2 Example: option parsers

To illustrate how the free applicative construction can be used in practice, we take as a running example a parser for options of a command-line tool.
For simplicity, we will limit ourselves to an interface which can only accept options that take a single argument. We will use a double dash as a prefix for the option name.
For example, a tool to create a new user in a Unix system could be used as follows:

```
create_user --username john \
            --fullname "John Doe" \
            --id 1002
```

Our parser could be run over the argument list and it would return a record of the following type:

```
data User = User
  { username :: String
  , fullname :: String
  , id :: Int}
  deriving Show
```

Furthermore, given a parser, it should be possible to automatically produce a summary of all the options that it supports, to be presented to the user of the tool as documentation.
We can define a data structure representing a parser for an individual option, with a specified type, as a functor:

```
data Option a = Option
  { optName :: String
  , optDefault :: Maybe a
  , optReader :: String → Maybe a}
  deriving Functor
```

We now want to create a DSL based on the Option functor, which would allow us to combine options for different types into a single value representing the full parser. As stated in the introduction, a common way to create a DSL from a functor is to use free monads. However, taking the free monad over the Option functor would not be very useful here. First of all, sequencing of options should be *independent*: later options should not depend on the value parsed by previous ones. Secondly, monads cannot be inspected without running them, so there is no way to obtain a summary of all options of a parser automatically.
What we really need is a way to construct a parser DSL in such a way that the values returned by the individual options can be combined using an Applicative interface. And that's exactly what FreeA will provide.
Thus, if we use FreeA Option a as our embedded DSL, we can interpret it as the type of a parser with an unspecified number of options, of possibly different types. When run, those options would be matched against the input command line, in an arbitrary order, and the resulting values will be eventually combined to obtain a final result of type a.
In our specific example, an expression to specify the command line option parser for create_user would look like this:

```
userP :: FreeA Option User
userP = User
  <$> one (Option "username" Nothing Just)
  <*> one (Option "fullname" (Just "") Just)
  <*> one (Option "id" Nothing readInt)
readInt :: String → Maybe Int
```

where we need a "generic smart constructor":

```
one :: Option a → FreeA Option a
```

which lifts an option to a parser.

1.3 Example: limited IO

One of the applications of free monads, exemplified in [13], is the definition of special-purpose monads, allowing to express computations which make use of a limited and well-defined subset of IO operations.
Given the following functor:

```
data FileSystem a =
    Read FilePath (String → a)
  | Write FilePath String a
  deriving Functor
```

the free monad on FileSystem, once "smart constructors" are defined for the two basic operations of reading and writing, allows to express any operation on files with the same convenience as the IO monad.
For example, one can implement a *copy* operation as follows:

```
copy :: FilePath → FilePath → Free FileSystem ()
copy src dst = read src >>= write dst
```

For some applications, we might need to have more control over the operations that are going to be executed when we eventually run the embedded program contained in a value of type Free FileSystem a.
For example, it could be useful to print a summary of the files that are going to be overwritten, and how much data in total is going to be written to disk.
However, there is no way to do that using the free monad approach. For example, there is no function:

```
count :: Free FileSystem a → Int
```

which returns the number of read/write operations performed by a monadic value.
To see why, consider the following example:

```
ex :: Free FileSystem ()
ex = do
  s ← read "/etc/motd"
  when (null s) $
    write "/tmp/out" ""
```

Now, ex performs 1 operation if and only if /etc/motd is empty, which, of course, cannot be determined by a pure function like count.
The FreeA construction, presented in this paper, represents a general solution for the problem of constructing embedded languages that allow the definition of functions performing static analysis on embedded programs, of which count :: FreeA FileSystem a → Int is a very simple example.

1.4 Example: applicative parsers

The idea that monads are "too flexible" has also been explored, again in the context of parsing, by Swierstra and Duponcheel ([12]), who showed how to improve both performance and error-reporting capabilities of an embedded language for grammars by giving up some of the expressivity of monads.
The basic principle is that, by weakening the monadic interface to that of an applicative functor (or, more precisely, an *alternative* functor), it becomes possible to perform enough static analysis to compute first sets for productions.
The approach followed in [12] is ad-hoc: an applicative functor is defined, which keeps track of first sets, and whether a parser accepts

the empty string. This is combined with a traditional monadic parser, regarded as an applicative functor, using a generalized semi-direct product, as described in [10].

The question, then, is whether it is possible to express this construction in a general form, in such a way that, given a functor representing a notion of "parser" for an individual symbol in the input stream, applying the construction one would automatically get an Applicative functor, allowing such elementary parsers to be sequenced.

Free applicative functors can be used to that end. We start with a functor `f`, such that `f a` describes an elementary parser for individual elements of the input, returning values of type `a`. `FreeA f a` is then a parser which can be used on the full input, and combines all the outputs of the individual parsers out of which it is built, yielding a result of type `a`.

Unfortunately, applying this technique directly results in a strictly less expressive solution. In fact, since `FreeA f` is the simplest applicative over `f`, it is necessarily *just* and applicative, i.e. it cannot also have an `Alternative` instance, which in this case is essential. We discuss the issue of `Alternative` in more detail in section 10.

2. Definition

To obtain a suitable definition for the free applicative functor generated by a functor `f`, we first pause to reflect on how one could naturally arrive at the definition of the `Applicative` class via an obvious generalisation of the notion of functor.

Given a functor `f`, the `fmap` method gives us a way to lift *unary* pure functions `a → b` to effectful functions `f a → f b`, but what about functions of arbitrary arity?

For example, given a value of type `a`, we can regard it as a nullary pure function, which we might want to lift to a value of type `f a`.

Similarly, given a binary function `h :: a → b → c`, it is quite reasonable to ask for a lifting of `h` to something of type `f a → f b → f c`.

The `Functor` instance alone cannot provide either of such liftings, nor any of the higher-arity liftings which we could define.

It is therefore natural to define a type class for generalised functors, able to lift functions of arbitrary arity:

```
class Functor f ⇒ MultiFunctor f where
  fmap₀ :: a → f a
  fmap₁ :: (a → b) → f a → f b
  fmap₁ = fmap
  fmap₂ :: (a → b → c) → f a → f b → f c
```

It is easy to see that a higher-arity $\mathtt{fmap}_n$ can now be defined in terms of $\mathtt{fmap}_2$. For example, for $\mathtt{n} = 3$:

```
fmap₃ :: MultiFunctor f
      ⇒ (a → b → c → d)
      → f a → f b → f c → f d
fmap₃ h x y z = fmap₂ ($) (fmap₂ h x y) z
```

However, before trying to think of what the laws for such a type class ought to be, we can observe that `MultiFunctor` is actually none other than `Applicative` in disguise.

In fact, $\mathtt{fmap}_0$ has exactly the same type as `pure`, and we can easily convert $\mathtt{fmap}_2$ to `(<*>)` and vice versa:

```
g <*> x = fmap₂ ($) g x
fmap₂ h x y = fmap h x <*> y
```

The difference between `(<*>)` and $\mathtt{fmap}_2$ is that `(<*>)` expects the first two arguments of $\mathtt{fmap}_2$, of types `a → b → c` and `f a` respectively, to be combined in a single argument of type `f (b → c)`.

This can always be done with a single use of `fmap`, so, if we assume that `f` is a functor, `(<*>)` and $\mathtt{fmap}_2$ are effectively equivalent.

Nevertheless, this roundabout way of arriving to the definition of `Applicative` shows that an applicative functor is just a functor that knows how to lift functions of arbitrary arities. An overloaded notation to express the application of $\mathtt{fmap}_i$ for all i is defined in [9], where it is referred to as *idiom brackets*.

Given a pure function of arbitrary arity and effectful arguments:

$$
\begin{aligned}
&\mathtt{h} : \mathtt{b}_1 \to \mathtt{b}_2 \to \cdots \to \mathtt{b}_n \to \mathtt{a}\\
&\mathtt{x}_1 : \mathtt{f}\ \mathtt{b}_1\\
&\mathtt{x}_2 : \mathtt{f}\ \mathtt{b}_2\\
&\cdots\\
&\mathtt{x}_n : \mathtt{f}\ \mathtt{b}_n
\end{aligned}
$$

the idiom bracket notation is defined as:

$$[\![\, \mathtt{h}\ \mathtt{x}_1\ \mathtt{x}_2 \cdots \mathtt{x}_n \,]\!] = \mathtt{pure}\ \mathtt{h} \mathbin{\texttt{<*>}} \mathtt{x}_1 \mathbin{\texttt{<*>}} \mathtt{x}_2 \mathbin{\texttt{<*>}} \cdots \mathbin{\texttt{<*>}} \mathtt{x}_n$$

We can build such an expression formally by using a `PureL` constructor corresponding to `pure` and a left-associative infix `(:*:)` constructor corresponding to `(<*>)`:

$$\mathtt{PureL}\ \mathtt{h} \mathbin{\texttt{:*:}} \mathtt{x}_1 \mathbin{\texttt{:*:}} \mathtt{x}_2 \mathbin{\texttt{:*:}} \cdots \mathbin{\texttt{:*:}} \mathtt{x}_n$$

The corresponding inductive definition is:

```
data FreeAL f a
  = PureL a
  | ∀b.FreeAL f (b → a) :*: f b
infixl 4 :*:
```

The `MultiFunctor` typeclass, the idiom brackets and the `FreeAL` definition correspond to the left parenthesised canonical form[1] of expressions built with `pure` and `(<*>)`. Just as lists built with concatenation have two canonical forms (cons-list and snoc-list) we can also define a right-parenthesised canonical form for applicative functors — a pure value over which a sequence of effectful functions are applied:

$$
\begin{aligned}
&\mathtt{x} : \mathtt{b}_1\\
&\mathtt{h}_1 : \mathtt{f}\ (\mathtt{b}_1 \to \mathtt{b}_2)\\
&\mathtt{h}_2 : \mathtt{f}\ (\mathtt{b}_2 \to \mathtt{b}_3)\\
&\cdots\\
&\mathtt{h}_n : \mathtt{f}\ (\mathtt{b}_n \to \mathtt{a})\\
&\mathtt{h}_n \mathbin{\texttt{<*>}} (\cdots \mathbin{\texttt{<*>}} (\mathtt{h}_2 \mathbin{\texttt{<*>}} (\mathtt{h}_1 \mathbin{\texttt{<*>}} \mathtt{pure}\ \mathtt{x})) \cdots)
\end{aligned}
$$

Replacing `pure` with a constructor `Pure` and `(<*>)` by a right-associative infix `(:$:)` constructor gives the following expression:

$$\mathtt{h}_n \mathbin{\texttt{:\$:}} \cdots \mathbin{\texttt{:\$:}} \mathtt{h}_2 \mathbin{\texttt{:\$:}} \mathtt{h}_1 \mathbin{\texttt{:\$:}} \mathtt{Pure}\ \mathtt{x}$$

The corresponding inductive type:

```
data FreeA f a
  = Pure a
  | ∀b.f (b → a) :$: FreeA f b
infixr 4 :$:
```

`FreeAL` and `FreeA` are isomorphic (see section 5), we pick the right-parenthesised version as our official definition since it is simpler to define the `Functor` and `Applicative` instances:

```
instance Functor f ⇒ Functor (FreeA f) where
  fmap g (Pure x)   = Pure (g x)
  fmap g (h :$: x) = fmap (g ◦) h :$: x
```

[1] Sometimes called simplified form because it is not necessarily unique.

The functor laws can be verified by structural induction, simply applying the definitions and using the functor laws for `f`.

```
instance Functor f ⇒ Applicative (FreeA f) where
  pure = Pure
  Pure g <*> y     = fmap g y
  (h :$: x) <*> y = fmap uncurry h :$:
                        ((,) <$> x <*> y)
```

In the last clause of the `Applicative` instance, `h` has type `f (x → y → z)`, and we need to return a value of type `FreeA f z`. Since `( :$: )` only allows us to express applications of 1-argument "functions", we uncurry `h` to get a value of type `f ((x , y) → z)`, then we use `( <*> )` recursively (see section 8 for a justification of this recursive call) to pair `x` and `y` into a value of type `FreeA f (x , y)`, and finally use the `( :$: )` constructor to build the result. Note the analogy between the definition of `( <*> )` and `( ++ )` for lists.

3. Applications

3.1 Example: option parsers (continued)

By using our definition of free applicative, we can compose the command line option parser exactly as shown in section 1.2 in the definition of `userP`. The smart constructor `one` which lifts an option (a functor representing a basic operation of our embedded language) to a term in our language can now be implemented as follows:

```
one :: Option a → FreeA Option a
one opt = fmap const opt :$: Pure ()
```

In section 7 we generalize `one` to any functor and by using generic functions specified as part of the adjunction we define functions which make use of the fact that it is possible to statically analyse a parser definition: functions are given for listing all possible options and for parsing a list of command line arguments given in arbitrary order.

3.2 Example: limited IO (continued)

In section 1.3 we showed an embedded DSL for file system operations based on free monads does not support certain kinds of static analysis.
However, we can now remedy this by using a free applicative, over the same functor `FileSystem`. In fact, the `count` function is now definable for `FreeA FileSystem a`. Moreover, this is not limited to this particular example: it is possible to define `count` for the free applicative over *any* functor.

```
count :: FreeA f a → Int
count (Pure _) = 0
count (_ :$: u) = 1 + count u
```

Of course, the extra power comes at a cost. Namely, the expressivity of the corresponding embedded language is severely reduced.
Using `FreeA FileSystem`, the files on which read and write operations are performed must be known in advance, as well as the content that is going to be written.
In particular, what one writes to a file cannot depend on what has been previously read, so operations like `copy` cannot be implemented.

3.3 Summary of examples

Applicative functors are useful for describing certain kinds of effectful computations. The free applicative construct over a given functor specifying the "basic operations" of an embedded language gives rise to terms of the embedded DSL built by applicative operators. These terms are only capable of representing a certain kind of effectful computation which can be described best with the help of the left-parenthesised canonical form: a pure function applied to effectful arguments. The calculation of the arguments may involve effects but in the end the arguments are composed by a pure function, which means that the effects performed are fixed when specifying the applicative expression.
In the case of the option parser example `userP`, the pure function is given by the `User` constructor and the "basic operation" `Option` is defining an option. The effects performed depend on how an evaluator is defined over an expression of type `FreeA Option a` and the order of effects can depend on the implementation of the evaluator.
For example, if one defines an embedded language for querying a database, and constructs applicative expressions using `FreeA`, one might analyze the applicative expression and collect information on the individual database queries by defining functions similar to the `count` function in the limited IO example. Then, different, possibly expensive duplicate queries can be merged and performed at once instead of executing the effectful computations one by one. By restricting the expressivity of our language we gain freedom in defining how the evaluator works.
One might define parts of an expression in an embedded DSL using the usual free monad construction, other parts using `FreeA` and compose them by lifting the free applicative expression to the free monad using the following function:

```
liftA2M :: Functor f ⇒ FreeA f a → Free f a
liftA2M (Pure x) = Return x
liftA2M (h :$: x) = Free
  (fmap (λf → fmap f (liftA2M x)) h)
```

In the parts of the expression defined using the free monad construction, the order of effects is fixed and the effects performed can depend on the result of previous effectful computations, while the free applicative parts have a fixed structure with effects not depending on each other. The monadic parts of the computation can depend on the result of static analysis carried out over the applicative part:

```
test :: FreeA FileSystem Int → Free FileSystem ()
test op = do
   ...
  let n = count op    -- result of static analysis
  n′ ← liftA2M op     -- result of applicative computation
  max ← read "max"
  when (max ⩾ n + n′) $ write "/tmp/test" "blah"
   ...
```

The possibility of using the results of static analysis instead of the need of specifying them by hand (in our example, this would account to counting certain function calls in an expression by looking at the code) can make the program less redundant.

4. Parametricity

In order to prove anything about our free applicative construction, we need to make an important observation about its definition.
The `( :$: )` constructor is defined using an existential type `b`, and it is clear intuitively that there is no way, given a value of the form `g :$: x`, to make use of the type `b` hidden in it.
More specifically, any function on `FreeA f a` must be defined *polymorphically* over all possible types `b` which could be used for the existentially quantified variable in the definition of `( :$: )`.
To make this intuition precise, we appeal to the notion of *relational parametricity* ([11], [14]), which, specialised to the `( :$: )` constructor, implies that:

```
( :$: ) :: ∀b.f (b → a) → (FreeA f b → FreeA f a)
```

is a natural transformation of contravariant functors. The two contravariant functors here could be defined, in Haskell, using a **newtype**:

```
newtype F1 f a x = F1 (f (x → a))
newtype F2 f a x = F2 (FreeA f x → FreeA f a)

instance Functor f ⇒ Contravariant (F1 f a) where
  contramap h (F1 g) = F1 $ fmap (∘ h) g

instance Functor f ⇒ Contravariant (F2 f a) where
  contramap h (F2 g) = F2 $ g ∘ fmap h
```

The action of F1 and F2 on morphisms is defined in the obvious way. Note that here we make use of the fact that FreeA f is a functor.

Naturality of (:$:) means that, given types x and y, and a function $h : x \to y$, the following holds:

$$\forall g :: f\ (y \to a), u :: \mathtt{FreeA}\ f\ x.$$
$$\mathtt{fmap}\ (\circ\, h)\ g \mathbin{:\$:} u \equiv g \mathbin{:\$:} \mathtt{fmap}\ h\ u \tag{1}$$

where we have unfolded the definitions of contramap for F1 and F2, and removed the newtypes.

5. Isomorphism of the two definitions

In this section we show that the two definitions of free applicatives given in section 2 are isomorphic.

First of all, if f is a functor, FreeAL f is also a functor:

```
instance Functor f ⇒ Functor (FreeAL f) where
  fmap g (PureL x) = PureL (g x)
  fmap g (h :*: x)  = (fmap (g ∘ ) h) :*: x
```

Again, the functor laws can be verified by a simple structural induction.

For the (:*:) constructor, a free theorem can be derived in a completely analogous way to deriving equation 1. This equation states that (:*:) is a natural transformation:

$$\forall h :: x \to y, g :: \mathtt{FreeAL}\ f\ (y \to a), u :: f\ x.$$
$$\mathtt{fmap}\ (\circ\, h)\ g \mathbin{:*:} u \equiv g \mathbin{:*:} \mathtt{fmap}\ h\ u \tag{2}$$

We define functions to convert between the two definitions:

```
r2l :: Functor f ⇒ FreeA f a FreeAL f a
r2l (Pure x) = PureL x
r2l (h :$: x) = fmap (flip ($)) (r2l x) :*: h

l2r :: Functor f ⇒ FreeAL f a FreeA f a
l2r (PureL x) = Pure x
l2r (h :*: x) = fmap (flip ($)) x :$: l2r h
```

We will also need the fact that l2r is a natural transformation:

$$\forall h :: x \to y, u :: \mathtt{FreeAL}\ f\ x.$$
$$\mathtt{l2r}\ (\mathtt{fmap}\ h\ u) \equiv \mathtt{fmap}\ h\ (\mathtt{l2r}\ u) \tag{3}$$

Proposition 1. r2l *is an isomorphism, the inverse of which is* l2r.

Proof. First we prove that $\forall u :: \mathtt{FreeA}\ f\ a.\mathtt{l2r}\ (\mathtt{r2l}\ u) \equiv u$. We compute using equational reasoning with induction on u:

```
  l2r (r2l (Pure x))
≡ ⟨ definition of r2l ⟩
  l2r (PureL x)
≡ ⟨ definition of l2r ⟩
  Pure x

  l2r (r2l (h :$: x))
≡ ⟨ definition of r2l ⟩
  l2r (fmap (flip ($)) (r2l x) :*: h)
≡ ⟨ definition of l2r ⟩
  fmap (flip ($)) h :$:
  l2r (fmap (flip ($)) (r2l x))
≡ ⟨ equation 3 ⟩
  fmap (flip ($)) h :$:
  fmap (flip ($)) (l2r (r2l x))
≡ ⟨ inductive hypothesis ⟩
  fmap (flip ($)) h :$: fmap (flip ($)) x
≡ ⟨ equation 1 ⟩
  fmap ( ∘ (flip ($))) (fmap (flip ($)) h) :$: x
≡ ⟨ f is a functor ⟩
  fmap (( ∘ (flip ($))) ∘ flip ($)) h :$: x
≡ ⟨ definition of flip and ($) ⟩
  fmap id h :$: x
≡ ⟨ f is a functor ⟩
  h :$: x
```

Next, we prove that $\forall u :: \mathtt{FreeAL}\ f\ a.\mathtt{r2l}\ (\mathtt{l2r}\ u) \equiv u$. Again, we compute using equational reasoning with induction on u:

```
  r2l (l2r (PureL x))
≡ ⟨ definition of l2r ⟩
  r2l (Pure x)
≡ ⟨ definition of r2l ⟩
  PureL x

  r2l (l2r (h :*: x))
≡ ⟨ definition of l2r ⟩
  r2l (fmap (flip ($)) x :$: l2r h)
≡ ⟨ definition of r2l ⟩
  fmap (flip ($)) (r2l (l2r h)) :*: fmap (flip ($)) x
≡ ⟨ inductive hypothesis ⟩
  fmap (flip ($)) h :*: fmap (flip ($)) x
≡ ⟨ equation 2 ⟩
  fmap ( ∘ (flip ($))) (fmap (flip ($)) h) :*: x
≡ ⟨ FreeAL f is a functor ⟩
  fmap (( ∘ (flip ($))) ∘ flip ($)) h :*: x
≡ ⟨ definition of flip and ($) ⟩
  fmap id h :*: x
≡ ⟨ FreeAL f is a functor ⟩
  h :*: x
```

□

In the next sections, we will prove that FreeA is a free applicative functor. Because of the isomorphism of the two definitions, these results will carry over to FreeAL.

6. Applicative laws

Following [9], the laws for an Applicative instance are:

$$\mathtt{pure\ id} \mathbin{<\!*\!>} u \equiv u \tag{4}$$
$$\mathtt{pure}\ (\circ) \mathbin{<\!*\!>} u \mathbin{<\!*\!>} v \mathbin{<\!*\!>} x \equiv u \mathbin{<\!*\!>} (v \mathbin{<\!*\!>} x) \tag{5}$$
$$\mathtt{pure}\ f \mathbin{<\!*\!>} \mathtt{pure}\ x \equiv \mathtt{pure}\ (f\ x) \tag{6}$$
$$u \mathbin{<\!*\!>} \mathtt{pure}\ x \equiv \mathtt{pure}\ (\$\, x) \mathbin{<\!*\!>} u \tag{7}$$

We introduce a few abbreviations to help make the notation lighter:

```
uc = uncurry
pair x y = ( , ) <$> x <*> y
```

Lemma 1. *For all*

```
u :: y → z
v :: FreeA f (x → y)
x :: FreeA f x
```

the following equation holds:

```
fmap u (v <*> x) ≡ fmap (u ∘ ) v <*> x
```

Proof. We compute:

```
  fmap u (Pure v <*> x)
≡ ⟨ definition of ( <*> ) ⟩
  fmap u (fmap v x)
≡ ⟨ FreeA f is a functor ⟩
  fmap (u ∘ v) x
≡ ⟨ definition of ( <*> ) ⟩
  Pure (u ∘ v) <*> x
≡ ⟨ definition of fmap ⟩
  fmap (u ∘ ) (Pure v) <*> x
```

```
  fmap u ((g :$: y) <*> x)
≡ ⟨ definition of ( <*> ) ⟩
  fmap u (fmap uc g :$: pair y x)
≡ ⟨ definition of fmap ⟩
  fmap (u ∘ ) (fmap uc g) :$: pair y x
≡ ⟨ f is a functor ⟩
  fmap (λg → u ∘ uc g) g :$: pair y x
≡ ⟨ f is a functor ⟩
  fmap uc (fmap ((u ∘ ) ∘ ) g) :$: pair y x
≡ ⟨ definition of ( <*> ) ⟩
  (fmap ((u ∘ ) ∘ ) g :$: y) <*> x
≡ ⟨ definition of fmap ⟩
  fmap (u ∘ ) (g :$: y) <*> x
```

□

Lemma 2. *Property 5 holds for* FreeA f*, i.e. for all*

```
u :: FreeA f (y → z)
v :: FreeA f (x → y)
x :: FreeA f x,
```

```
pure ( ∘ ) <*> u <*> v <*> x ≡ u <*> (v <*> x)
```

Proof. Suppose first that $u = \mathtt{Pure}\ u_0$ for some $u_0 :: y \to z$:

```
  Pure ( ∘ ) <*> Pure u₀ <*> v <*> x
≡ ⟨ definition of ( <*> ) ⟩
  Pure (u₀ ∘ ) <*> v <*> x
≡ ⟨ definition of ( <*> ) ⟩
  fmap (u₀ ∘ ) v <*> x
≡ ⟨ lemma 1 ⟩
  fmap u₀ (v <*> x)
≡ ⟨ definition of ( <*> ) ⟩
  Pure u₀ <*> (v <*> x)
```

To tackle the case where u = g :$: w, for

```
g :: f (w → y → z)
w :: FreeA f w,
```

we need to define a helper function

```
t :: ((w , x → y) , x) → (w , y)
t ((w , v) , x) = (w , v x)
```

and compute:

```
  pure ( ∘ ) <*> (g :$: w) <*> v <*> x
≡ ⟨ definition of pure and ( <*> ) ⟩
  (fmap (( ∘ ) ∘ ) g :$: w) <*> v <*> x
≡ ⟨ definition of composition ⟩
  (fmap (λg w v → g w ∘ v) g :$: w) <*> v <*> x
≡ ⟨ definition of ( <*> ) ⟩
  (fmap uc (fmap (λg w v → g w ∘ v) g) :$: pair w v)
  <*> x
≡ ⟨ f is a functor and definition of uc ⟩
  (fmap (λg (w , v) → g w ∘ v) g :$: pair w v) <*> x
≡ ⟨ definition of ( <*> ) ⟩
  fmap uc (fmap (λg (w , v) → g w ∘ v) g) :$:
  pair (pair w v) x
≡ ⟨ f is a functor and definition of uc ⟩
  fmap (λg ((w , v) , x) → g w (v x)) g :$:
  pair (pair w v) x
≡ ⟨ definition of uc and t ⟩
  fmap (λg → uc g ∘ t) g :$: pair (pair w v) x
≡ ⟨ f is a functor ⟩
  fmap ( ∘ t) (fmap uc g) :$: pair (pair w v) x
≡ ⟨ equation 1 ⟩
  fmap uc g :$: fmap t (pair (pair w v) x)
≡ ⟨ lemma 1 (3 times) and FreeA f is a functor (3 times) ⟩
  fmap uc g :$: (pure ( ∘ ) <*> fmap ( , ) w <*> v <*> x)
≡ ⟨ induction hypothesis for fmap ( , ) w ⟩
  fmap uc g :$: (fmap ( , ) w <*> (v <*> x))
≡ ⟨ definition of ( <*> ) ⟩
  (g :$: w) <*> (v <*> x)
```

□

Lemma 3. *Property 7 holds for* FreeA f*, i.e. for all*

```
u :: FreeA f (x → y)
x :: x,
```

```
u <*> pure x ≡ pure ( $ x) <*> u
```

Proof. If u is of the form Pure u_0, then the conclusion follows immediately.
Let's assume, therefore, that u = g :$: w, for some w::w, g::f (w → x → y), and that the lemma is true for structurally smaller values of u:

```
  (g :$: w) <*> pure x
≡ ⟨ definition of ( <*> ) ⟩
  fmap uc g :$: pair w (pure x)
≡ ⟨ definition of pair ⟩
  fmap uc g :$: (fmap ( , ) w <*> pure x)
≡ ⟨ induction hypothesis for fmap ( , ) w ⟩
  fmap uc g :$: (pure ( $ x) <*> fmap ( , ) w)
≡ ⟨ FreeA f is a functor ⟩
  fmap uc g :$: fmap (λw → (w , x)) w)
≡ ⟨ equation 1 ⟩
  fmap (λg w → g (w , x)) (fmap uc g) :$: w
≡ ⟨ f is a functor ⟩
  fmap (λg w → g w x) g :$: w
≡ ⟨ definition of fmap for FreeA f ⟩
  fmap ( $ x) (g :$: w)
≡ ⟨ definition of ( <*> ) ⟩
  pure ( $ x) <*> (g :$: w)
```

□

Proposition 2. FreeA f *is an applicative functor.*

Proof. Properties 4 and 6 are straightforward to verify using the fact that `FreeA f` is a functor, while properties 5 and 7 follow from lemmas 2 and 3 respectively. □

7. `FreeA` as a Left adjoint

We now want to show that `FreeA f` is really the free applicative functor on `f`. For that, we need to define a category of applicative functors $\mathcal{A}$, and show that `FreeA` is a functor

$$\texttt{FreeA} : \mathcal{F} \to \mathcal{A},$$

where $\mathcal{F}$ is the category of endofunctors of `Hask`, and that `FreeA` is left adjoint to the forgetful functor $\mathcal{A} \to \mathcal{F}$.

Definition 1. *Let* `f` *and* `g` *be two applicative functors. An applicative natural transformation between* `f` *and* `g` *is a polymorphic function*

$$\texttt{t} :: \forall \texttt{a}.\texttt{f a} \to \texttt{g a}$$

satisfying the following laws:

$$\texttt{t (pure x)} \equiv \texttt{pure x} \qquad (8)$$

$$\texttt{t (h <*> x)} \equiv \texttt{t h <*> t x}. \qquad (9)$$

We define the type of all applicative natural transformations between `f` and `g`, we write, in Haskell,

```
type AppNat f g = ∀a.f a → g a
```

where the laws are implied.
Similarly, for any pair of functors `f` and `g`, we define

```
type Nat f g = ∀a.f a → g a
```

for the type of natural transformations between `f` and `g`.
Note that, by parametricity, polymorphic functions are automatically natural transformations in the categorical sense, i.e, for all

$$\texttt{t} :: \texttt{Nat f g}$$
$$\texttt{h} :: \texttt{a} \to \texttt{b}$$
$$\texttt{x} :: \texttt{f a},$$

$$\texttt{t (fmap h x)} \equiv \texttt{fmap h (t x)}.$$

It is clear that applicative functors, together with applicative natural transformations, form a category, which we denote by $\mathcal{A}$, and similarly, functors and natural transformations form a category $\mathcal{F}$.

Proposition 3. `FreeA` *defines a functor* $\mathcal{F} \to \mathcal{A}$.

Proof. We already showed that `FreeA` sends objects (functors in our case) to applicative functors.
We need to define the action of `FreeA` on morphisms (which are natural transformations in our case):

```
liftT :: (Functor f, Functor g)
      ⇒ Nat f g
      → AppNat (FreeA f) (FreeA g)
liftT _ (Pure x)  = Pure x
liftT k (h :$: x) = k h :$: liftT k x
```

First we verify that `liftT k` is an applicative natural transformation i.e. it satisfies laws 8 and 9. We use equational reasoning for proving law 8:

```
  liftT k (pure x)
≡ ⟨ definition of pure ⟩
  liftT k (Pure x)
≡ ⟨ definition of liftT ⟩
  Pure x
≡ ⟨ definition of pure ⟩
  pure x
```

For law 9 we use induction on the size of the first argument of `(<*>)` as explained in section 8. The base cases:

```
  liftT k (Pure h <*> Pure x)
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap h (Pure x))
≡ ⟨ definition of fmap ⟩
  liftT k (Pure (h x))
≡ ⟨ definition of liftT ⟩
  Pure (h x)
≡ ⟨ definition of fmap ⟩
  fmap h (Pure x)
≡ ⟨ definition of (<*>) ⟩
  Pure h <*> Pure x
≡ ⟨ definition of liftT ⟩
  liftT k (Pure h) <*> liftT k (Pure x)
```

```
  liftT k (Pure h <*> (i :$: x))
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap h (i :$: x))
≡ ⟨ definition of fmap ⟩
  liftT k (fmap (h ∘) i :$: x)
≡ ⟨ definition of liftT ⟩
  k (fmap (h ∘) i) :$: liftT k x
≡ ⟨ k is natural ⟩
  fmap (h ∘) (k i) :$: liftT k x
≡ ⟨ definition of fmap ⟩
  fmap h (k i :$: liftT k x)
≡ ⟨ definition of (<*>) ⟩
  Pure h <*> (k i :$: liftT k x)
≡ ⟨ definition of liftT ⟩
  liftT k (Pure h) <*> liftT k (i :$: x)
```

The inductive case:

```
  liftT k ((h :$: x) <*> y)
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap uncurry h :$: (fmap (,) x <*> y)
≡ ⟨ definition of liftT ⟩
  k (fmap uncurry h) :$: liftT k (fmap (,) x <*> y)
≡ ⟨ inductive hypothesis ⟩
  k (fmap uncurry h) :$:
  (liftT k (fmap (,) x) <*> liftT k y)
≡ ⟨ liftT k is natural ⟩
  k (fmap uncurry h) :$:
  (fmap (,) (liftT k x) <*> liftT k y)
≡ ⟨ k is natural ⟩
  fmap uncurry (k h) :$:
  (fmap (,) (liftT k x) <*> liftT k y)
≡ ⟨ definition of (<*>) ⟩
  (k h :$: liftT k x) <*> liftT k y
≡ ⟨ definition of liftT ⟩
  liftT k (h :$: x) <*> liftT k y
```

Now we need to verify that `liftT` satisfies the functor laws

$$\texttt{liftT id} \equiv \texttt{id}$$

$$\texttt{liftT (t} \circ \texttt{u)} \equiv \texttt{liftT t} \circ \texttt{liftT u}.$$

The proof is a straightforward structural induction. □

We are going to need the following natural transformation:

```
one :: Functor f ⇒ Nat f (FreeA f)
one x = fmap const x :$: Pure ()
```

which embeds any functor `f` into `FreeA f` (we used a specialization of this function for `Option` in section 1.2).

Lemma 4.

$$\texttt{g :\$: x} \equiv \texttt{one g <*> x}$$

Proof. Given

```
h :: a → ((), a)
h x = ((), x)
```

it is easy to verify that:

$$(\circ\,\texttt{h}) \circ \texttt{uncurry} \circ \texttt{const} \equiv \texttt{id}, \qquad (10)$$

so

```
  one g <*> x
≡ ⟨ definition of one ⟩
  (fmap const g :$: Pure ()) <*> x
≡ ⟨ definition of (<*>) and functor law for f ⟩
  fmap (uncurry ∘ const) g :$: fmap h x
≡ ⟨ equation 1 and functor law for f ⟩
  fmap ((∘ h) ∘ uncurry ∘ const) g :$: x
≡ ⟨ equation 10 ⟩
  g :$: x
```

□

Proposition 4. *The* `FreeA` *functor is left adjoint to the forgetful functor* $\mathcal{A} \to \mathcal{F}$*. Graphically:*

$$Hom_{\mathcal{F}}(\texttt{FreeA f}, \texttt{g}) \underset{\overleftarrow{\texttt{raise}}}{\overset{\overrightarrow{\texttt{lower}}}{\cong}} Hom_{\mathcal{A}}(\texttt{f}, \texttt{g})$$

Proof. Given a functor `f` and an applicative functor `g`, we define a natural bijection between `Nat f g` and `AppNat (FreeA f) g` as such:

```
raise :: (Functor f, Applicative g)
      ⇒ Nat f g
      → AppNat (FreeA f) g
raise _ (Pure x) = pure x
raise k (g :$: x) = k g <*> raise k x

lower :: (Functor f, Applicative g)
      ⇒ AppNat (FreeA f) g
      → Nat f g
lower k = k ∘ one
```

A routine verification shows that `raise` and `lower` are natural in `f` and `g`. The proof that `raise k` satisfies the applicative natural transformation laws 8 and 9 is a straightforward induction having the same structure as the proof that `liftT k` satisfies these laws (proposition 3). To show that `f` and `g` are inverses of each other, we reason by induction and calculate in one direction:

```
  raise (lower t) (Pure x)
≡ ⟨ definition of raise ⟩
  pure x
≡ ⟨ t is an applicative natural transformation ⟩
  t (pure x)
≡ ⟨ definition of pure ⟩
  t (Pure x)

  raise (lower t) (g :$: x)
≡ ⟨ definition of raise ⟩
  lower t g <*> raise (lower t) x
≡ ⟨ induction hypothesis ⟩
  lower t g <*> t x
≡ ⟨ definition of lower ⟩
  t (one g) <*> t x
≡ ⟨ t is an applicative natural transformation ⟩
  t (one g <*> x)
≡ ⟨ lemma 4 ⟩
  t (g :$: x)
```

The other direction:

```
  lower (raise t) x
≡ ⟨ definition of lower ⟩
  raise t (one x)
≡ ⟨ definition of one ⟩
  raise t (fmap const x :$: Pure ())
≡ ⟨ definition of raise ⟩
  t (fmap const x) <*> pure ()
≡ ⟨ t is natural ⟩
  fmap const (t x) <*> pure ()
≡ ⟨ fmap h ≡ ((pure h) <*>) in an applicative functor ⟩
  pure const <*> t x <*> pure ()
≡ ⟨ t is natural ⟩
  pure ($ ()) <*> (pure const <*> t x)
≡ ⟨ applicative law 5 ⟩
  pure (∘) <*> pure ($ ()) <*> pure const <*> t x
≡ ⟨ applicative law 6 applied twice ⟩
  pure id <*> t x
≡ ⟨ applicative law 4 ⟩
  t x
```

□

7.1 Example: option parsers (continued)

With the help of the adjunction defined above by `raise` and `lower` we are able to define some useful functions. In the case of command-line option parsers, for example, it can be used for computing the global default value of a parser:

```
parserDefault :: FreeA Option a → Maybe a
parserDefault = raise optDefault
```

or for extracting the list of all the options in a parser:

```
allOptions :: FreeA Option a → [String]
allOptions = getConst ∘ raise f
  where
    f opt = Const [optName opt]
```

`allOptions` works by first defining a function that takes an option and returns a one-element list with the name of the option, and then lifting it to the `Const` applicative functor.
Using `parserDefault`, we can now write a function that runs an applicative option parser over a list of command-line arguments, accepting them in any order:

```
matchOpt :: String → String
         → FreeA Option a
         → Maybe (FreeA Option a)
matchOpt _ _ (Pure _) = Nothing
matchOpt opt value (g :$: x)
  | opt ≡ '-' : '-' : optName g
  = fmap (<$> x) (optReader g value)
  | otherwise
  = fmap (g :$:) (matchOpt opt value x)

runParser :: FreeA Option a
          → [String]
          → Maybe a
runParser p (opt : value : args) =
```

```
    case matchOpt opt value p of
      Nothing → Nothing
      Just p′ → runParser p′ args
runParser p [] = parserDefault p
runParser _ _ = Nothing
```

The `matchOpt` function looks for options in the parser which match the given command-line argument, and, if successful, returns a modified parser where the option has been replaced by a pure value. Finally, `runParser` calls `matchOpt` with successive pairs of arguments, until no arguments remain, at which point it uses the default values of the remaining options to construct a result.

8. Totality

All the proofs in this paper apply to a total fragment of Haskell, and completely ignore the presence of bottom.
To justify the validity of our results, then, we need to ensure that all definitions are actually possible in a total language.
In fact, all our ADT definitions can be regarded as inductive fixpoints of strictly positive functors, and most of the function definitions use primitive recursion, so they are obviously terminating for all inputs. Furthermore, most proofs are carried out by structural induction.
One exception is the definition of `(<*>)`:

```
(h :$: x) <*> y = fmap uncurry h :$: ((,) <$> x <*> y)
```

which contains a recursive call where the first argument, namely `(,) <$> x`, is not structurally smaller than the original one (`h :$: x`). To prove that this function is nevertheless total, we introduce a notion of *size* for values of type `FreeA f a`:

```
size :: FreeA f a → ℕ
size (Pure _) = 0
size (_ :$: x) = 1 + size x
```

To conclude that the definition of `(<*>)` is indeed terminating, we just need to show that the size of argument for the recursive call is smaller than the size of the original argument, which is an immediate consequence of the following lemma.

Lemma 5. *For any function* `f :: a → b` *and* `u :: FreeA f a`*,*

```
size (fmap f u) ≡ size u
```

Proof. By induction:

```
  size (fmap f (Pure x))
≡ ⟨ definition of fmap ⟩
  size (Pure (f x))
≡ ⟨ definition of size ⟩
  0
≡ ⟨ definition of size ⟩
  size (Pure x)

  size (fmap f (g :$: x))
≡ ⟨ definition of fmap ⟩
  size (fmap (f ∘) g :$: x)
≡ ⟨ definition of size ⟩
  1 + size x
≡ ⟨ definition of size ⟩
  size (g :$: x)
```

□

In most of our proofs using induction we carry out induction on the size of the first argument of `(<*>)` where size is defined by the above `size` function.

9. Related work

The idea of free applicative functors is not entirely new. There have been a number of different definitions of free applicative functor over a given Haskell functor, but none of them includes a proof of the applicative laws.
The first author of this paper published a specific instance of applicative functors similar to our example shown in section 1.2 ([4]). The example was developed further in the Haskell package `optparse-applicative` [3].
Tom Ellis proposes a definition very similar to ours ([6]), but uses a separate inductive type for the case corresponding to our `(:$:)` constructor. He then observes that law 6 probably holds because of the existential quantification, but doesn't provide a proof. We solve this problem by deriving the necessary equation 1 as a "free theorem".
Gergő Érdi gives another similar definition ([5]), but his version presents some redundancies, and thus fails to obey the applicative laws. For example, `Pure id <*> x` can easily be distinguished from `x` using the `count` function defined above.
The `free` package on hackage ([1]) contains a definition essentially identical to our `FreeAL`, differing only in the order of arguments.
Another approach, which differs significantly from the one presented in the paper, underlies the definition contained in the `free-functors` package on hackage ([2]), and uses a Church-like encoding (and the `ConstraintKinds` GHC extension) to generalize the construction of a free `Applicative` to any superclass of `Functor`.
The idea is to use the fact that, if a functor T has a left adjoint F, then the monad $F \circ T$ is the codensity monad of T (i.e. the right Kan extension of T along itself). By taking T to be the forgetful functor $\mathcal{A} \to \mathcal{F}$, one can obtain a formula for F using the expression of a right Kan extension as an end.
One problem with this approach is that the applicative laws, which make up the definition of the category $\mathcal{A}$, are left implicit in the universal quantification used to represent the end.
In fact, specializing the code in `Data.Functor.HFree` to the `Applicative` constraint, we get:

```
data FreeA′ f a = FreeA′ {
  runFreeA :: ∀g.Applicative g
           ⇒ (∀x.f x → g x) → g a}
instance Functor f ⇒ Functor (FreeA′ f) where
  fmap h (FreeA′ t) = FreeA′ (fmap h ∘ t)
instance Functor f ⇒ Applicative (FreeA′ f) where
  pure x = FreeA′ (\_ → pure x)
  FreeA′ t1 <*> FreeA′ t2 =
    FreeA′ (λu → t1 u <*> t2 u)
```

Now, for law 4 to hold, for example, we need to prove that the term `λu → pure id <*> t u` is equal to `t`. This is strictly speaking false, as those terms can be distinguished by taking any functor with an `Applicative` instance that doesn't satisfy law 4, and as `t` a constant function returning a counter-example for it.
Intuitively, however, the laws should hold provided we never make use of invalid `Applicative` instances. To make this intuition precise, one would probably need to extend the language with quantification over equations, and prove a parametricity result for this extension.
Another problem of the Church encoding is that it is harder to use. In fact, the destructor `runFreeA` is essentially equivalent to our `raise` function, which can only be used to define *applicative* natural transformation. A function like `matchOpt` in section 7.1, which is not applicative, could not be defined over `FreeA′`.

10. Discussion and further work

We have presented a practical definition of free applicative functor over any Haskell functor, proved its properties, and showed some of its applications. As the examples in this paper show, free applicative functors solve certain problems very effectively, but their applicability is somewhat limited.

For example, applicative parsers usually need an `Alternative` instance as well, and the free applicative construction doesn't provide that. One possible direction for future work is trying to address this issue by modifying the construction to yield a free `Alternative` functor, instead.

Unfortunately, there is no satisfactory set of laws for alternative functors: if we simply define an alternative functor as a monoid object in $\mathcal{A}$, then many commonly used instances become invalid, like the one for `Maybe`.

Another direction is formalizing the proofs in this paper in a proof assistant, by embedding the total subset of Haskell under consideration into a type theory with dependent types.

Our attempts to replicate the proofs in Agda have failed, so far, because of subtle issues in the interplay between parametricity and the encoding of existentials with dependent sums.

In particular, equation 1 is inconsistent with a representation of the existential as a Σ type in the definition of `FreeA`. For example, terms like `const () :$: Pure 3` and `id :$: Pure ()` are equal by equation 1, but can obviously be distinguished using large elimination.

The problem seems to be related to the difference in predicativity between System F and Martin-Löf type theory. Using the approach in [7] to add parametricity to the theory, one obtains a statement which is not powerful enough to prove 1, as the constructor `(:$:)` has values in a type which resides in a higher universe.

To overcome this limitation, the theory needs to provide a notion of *weak existential*, that is, a Σ type without large elimination, which would resemble Haskell existentials more faithfully. Although it is possible to define weak existentials in an impredicative theory (e.g. Coq with `--impredicative-set`) using a Church encoding, it is not clear how to do the same in predicative Martin-Löf type theory.

Another possible further development of the results in this paper is trying to generalize the construction of a free applicative functor to endofunctors in any monoidal category. In this more general setting, applicative functors correspond to lax monoidal functors, and the construction of this paper can be regarded as a left Kan extension.

In fact, suppose $\mathcal{C}$ is a monoidal category with unit I and operation $\oplus$, and let f be an endofunctor of $\mathcal{C}$.

To remove any form of recursion from our definition of `FreeA`, we consider the comonad on the category of monoidal categories MonCat corresponding to the adjunction between the forgetful functor to Cat and the functor giving the free monoidal category:

$$\begin{array}{ccc} \mathrm{MonCat} & \longrightarrow & \mathrm{MonCat} \\ C & \mapsto & C^* \end{array}$$

with counit $\epsilon : \mathcal{C}^* \to \mathcal{C}$.

If we interpret the existential as a coend, we get a very concise definition for `FreeA f`, as the left Kan extension of $\epsilon \circ f^*$ along ϵ.

The result of this paper could then be extended to this more general context, and possibly even further, by replacing monoidal categories with an arbitrary doctrine.

Acknowledgments

We would like to thank Jennifer Hackett, Thorsten Altenkirch, Venanzio Capretta, Graham Hutton, Edkso de Vries and Christian Sattler, for helpful suggestions and insightful discussions on the topics presented in this paper.

References

[1] `http://hackage.haskell.org/package/free`, .

[2] `http://hackage.haskell.org/package/free-functors`, .

[3] `http://hackage.haskell.org/package/optparse-applicative`.

[4] `http://paolocapriotti.com/blog/2012/04/27/applicative-option-parser`.

[5] `http://gergo.erdi.hu/blog/2012-12-01-static_analysis_with_applicatives/`.

[6] `http://web.jaguarpaw.co.uk/~tom/blog/posts/2012-09-09-towards-free-applicatives.html`.

[7] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.

[8] S. Marlow. Haskell 2010 language report, 2010.

[9] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968. doi: 10.1017/S0956796807006326. URL `http://dx.doi.org/10.1017/S0956796807006326`.

[10] R. Paterson. Constructing applicative functors. In *MPC*, pages 300–323, 2012.

[11] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[12] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *ADVANCED FUNCTIONAL PROGRAMMING*, pages 184–207. Springer-Verlag, 1996.

[13] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

[14] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.